Quantitative Characterization of the Software Layer of a HW/SW Co-Designed Processor

José Cano*, Rakesh Kumar*, Aleksandar Brankovic†, Demos Pavlou‡,
Kyriakos Stavrou‡, Enric Gibert§, Alejandro Martínez¶ and Antonio González‖
*University of Edinburgh, UK †Intel ‡11pets
§Pharmacelera ¶ARM ‖Universitat Politècnica de Catalunya, Spain

***Abstract*—HW/SW co-designed processors currently have a renewed interest due to their capability to boost performance without running into the power and complexity walls. By employing a software layer that performs dynamic binary translation and applies aggressive optimizations through exploiting the runtime application behavior, these hybrid architectures provide better performance/watt. However, a poorly designed software layer can result in significant translation/optimization overheads that may offset its benefits. This work presents a detailed characterization of the software layer of a HW/SW co-designed processor using a variety of benchmark suites. We observe that the performance of the software layer is very sensitive to the characteristics of the emulated application with a variance of more than 50%. We also show that the interaction between the software layer and the emulated application, while sharing the microarchitectural resources, can have 0-20% impact on performance. Finally, we identify some key elements which should be further investigated to reduce the observed variations in performance. The paper provides critical insights to improve the software layer design.**

I. Introduction

Hardware/Software (HW/SW) co-designed processors are hybrid architectures that couple a software layer to the microarchitectural implementation of a processor. The software layer resides between the hardware and the operating system (Figure 1b), and dynamically translates and optimizes binaries from a guest Instruction Set Architecture (ISA) to the host ISA — we refer to the software layer as Translation Optimization Layer (TOL). These processors have some important advantages over traditional hardware-only systems (Figure 1a) such as the ability to support multiple guest ISAs, the exploitation of dynamic information at runtime (which can potentially improve performance and/or power consumption), etc.

These hybrid architectures have gained a great momentum over the past few years. Earlier examples include Transmeta products Crusoe [1], [2] and Efficeon [3], as well as research projects from IBM like DAISY [4] and BOA [5]. Currently, there is a renewed interest from both industry [6], [7] and academia [8], [9], [10], [11], [12], [13]. The fundamental idea behind all these systems is to have a simple host ISA to achieve design simplicity and energy efficiency. But at the same time, boost the performance by aggressively optimizing the guest ISA binaries through the software layer.

This work was done when: José Cano, Rakesh Kumar, and Aleksandar Brankovic were with UPC; Demos Pavlou, Kyriakos Stavrou, Enric Gibert, Alejandro Martínez, and Antonio González were with Intel labs.

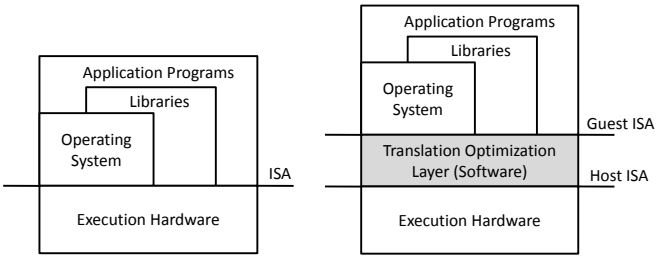


Fig. 1: HW/SW interface in processors: a) Conventional; b) Co-designed.

Although dynamic binary translation can also be performed in hardware, as in x86 processors, a software implementation provides several important benefits. For example, it allows to translate bigger code regions (or blocks of instructions) at once, thus increasing the scope of optimizations. Furthermore, the processor could be upgraded in the field by introducing new optimizations or fixing bugs in the software layer, which is not possible with the hardware implementation. As software is easier to develop and validate than hardware, validation and verification cost/time are also reduced. Finally, the co-design implementation reduces hardware complexity.

However, if the software layer is not carefully designed, it can also generate significant overheads that might compromise its benefits. We think it is of vital importance to understand those overheads. Nevertheless, to the best of our knowledge, no data have been made public that study in detail the characteristics of the software layer of a HW/SW co-designed processor, its interaction with the application, and how its execution profile is related to the application it emulates.

In this paper, we quantitatively characterize the software layer of a HW/SW co-designed processor using a variety of benchmark suites. The analysis provided shows the challenges that any co-design processor needs to address in order to be efficient. We make three important contributions:

- We study the performance of TOL with respect to the emulated application. We show that the particular characteristics of the application like the size of the static code, the dynamic/static instructions ratio, or the amount of indirect branches can affect TOL performance (and thus its overhead) with a variance of more than 50%.

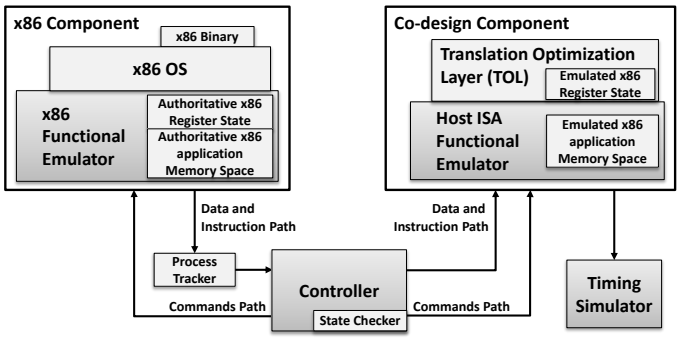


Fig. 2: DARCO main components.

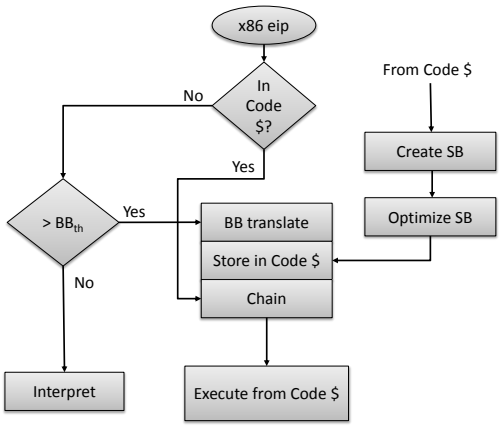


Fig. 3: Translation Optimization Layer (TOL) execution flow. The left path is followed in IM, the middle in BBM and the right in SBM.

- We examine how the execution time is affected by the interaction of TOL and the emulated application on shared microarchitectural resources. Due to this interaction, the performance can be penalized by up to 20%.
- We identify four key elements that should be further analyzed to improve TOL performance: data cache, instruction scheduling, instructions cache, and indirect branches.

II. Experimental methodology

In this section we briefly describe the infrastructure used to perform our characterization study, and our evaluation methodology along with the benchmarks suites analyzed.

A. Infrastructure description

We used DARCO [14], a simulation infrastructure specifically designed to enable research for HW/SW co-designed processors. DARCO models a HW/SW co-designed processor that executes guest x86 binaries on a RISC host architecture. The infrastructure consists of four main components (Figure 2): the *x86 Component*, the *Co-design Component*, the *Timing Simulator*, and the *Controller*.

- **x86 Component**: provides an x86 full-system functional emulator on top of which an unmodified operating system is executed. This component keeps the authoritative architectural state. Its main role is to permit co-simulation [15], a technique very useful for debugging which checks that the state of the translations/optimizations is correct according to the guest ISA semantics.
- **Co-design Component**: provides the host processor functional model. It executes the Translation Optimization Layer (TOL) that translates and optimizes the x86 instruction stream to simple RISC instructions.
- **Timing Simulator**: models a configurable RISC in-order processor. It receives the dynamic instruction stream from the co-design component and provides detailed execution statistics. The timing simulator is able to distinguish the instructions corresponding to the emulation of the x86 application from those corresponding to TOL. This feature enables studying the interaction between the x86 component and the co-design component at the microarchitectural level (Section III-D).
- **Controller**: is the main interface of DARCO with the user. It provides full control over the execution of the application, as well as debugging utilities.

Note that DARCO models only user-level code and skips system calls and exception handling routines. However, this has minor implications on the experimental results, mainly due to the small contribution of non user-level code to the dynamic code stream for the benchmarks studied. Next, we explain in deeper detail the two components that are relevant for this paper, that is, TOL and the timing simulator.

1) Translation Optimization Layer (TOL): Software layer that translates the guest code to the host ISA. It is equipped with an interpreter, a translator, a profiler and a dynamic optimizer that applies a plethora of optimizations to the translated regions. TOL has three different execution modes, interpretation mode (IM), basic block translation mode (BBM), and superblock and optimization mode (SBM). Figure 3 shows the high level view of the execution flow of TOL.

TOL starts by interpreting the x86 instructions. When a branch target executes more than a predefined threshold (IM/BBth), TOL switches to BBM, the particular basic block (BB) is translated and the translation is stored in the code cache. Subsequent executions of this BB are done from the code cache while gathering profiling information for the direction of the branch and its target, through instrumentation. When a BB executes more times than another predefined threshold (BB/SBth), it is considered hot and TOL switches to SBM. During this mode, the control flow profiling information that was collected during the previous executions is used by the optimizer in order to create a superblock (SB) with the starting point being the BB that triggered SBM. The SB passes through several optimizations (copy/constant propagation, constant folding, common subexpression elimination, dead code elimination, register allocation and instruction scheduling) and it is stored in the code cache. Subsequent executions of the SB are done from the code cache.

2) Timing Simulator: Models a simple in-order processor, following the philosophy of having a host machine as simple

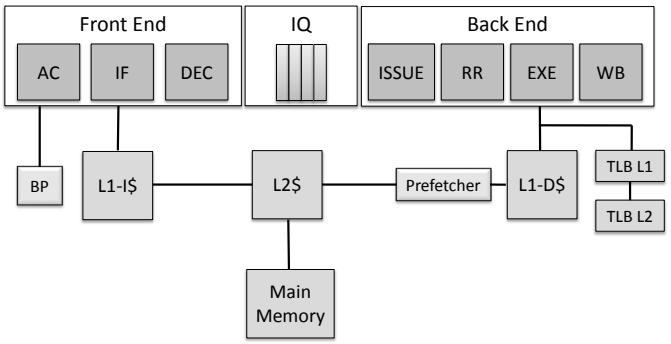


Fig. 4: Host processor model pipeline overview, composed of: Front-End, Instruction Queue (IQ), and Back-End.

as possible. The modeled pipeline decouples the Front-End from the Back-End using an Instruction Queue (Figure 4). The Front-End reads (Address Calculation (AC) stage, Instruction Fetch (IF) stage), decodes (DEC stage), and stores the instructions in the Instruction Queue (IQ). Furthermore, it is equipped with a Gshare branch predictor (BP) and a Branch Target Buffer (BTB). The Back-End issues and executes instructions from IQ. Issuing is done using a scoreboard that keeps track of the availability of the source registers. During the Register Read (RR) stage, the instructions read their operands from the bypass or the register file, which is logically divided between TOL and the application (32 registers are only accessible by TOL and 32 only by the translated application code) in order to reduce transition overheads. Cache accesses are performed during the EXE stage, which includes the address calculation. Also during the EXE stage, the processor executes the ALU operations. The last stage is the Write Back (WB). Notice that branch mispredictions are detected and handled in the EXE stage, leading to a misprediction penalty of 6 cycles.

The modeled processor has two levels of cache in the memory hierarchy. The first level is divided between instructions (L1-I$) and data (L1-D$), whereas the second level (L2) and main memory are shared. The translation lookaside buffer (TLB) has a two level architecture and exists only for data, since TOL works with physical addresses. Furthermore, notice that the Back-End is equipped with a stride prefetcher.

The processor configuration used across all the experiments is shown in Table I. The issue width of the processor is 2, with both pipes being symmetric. Each pipe has one simple (1 cycle latency) and one complex (2 cycles latency) integer execution units, and one simple (2 cycles latency) and one complex (5 cycles latency) floating point execution units.

B. Evaluation methodology

When evaluating a dynamic binary translator it is of key importance to distinguish between the start-up and the steady state. A non-strict definition of the start-up phase is that it is the amount of time spent interpreting and translating guest code until most of the code is translated and the frequency of translation is low. Similarly, a loose definition of the steady state is that it is the execution thereafter, with the majority of the guest code being executed from the code cache — note that this is the expected behavior for the common case. Note that there will always be applications either with big code footprint and low repetition or with very small number of dynamic instructions that will never reach the steady state.

TABLE I: Host processor microarchitectutal parameters.

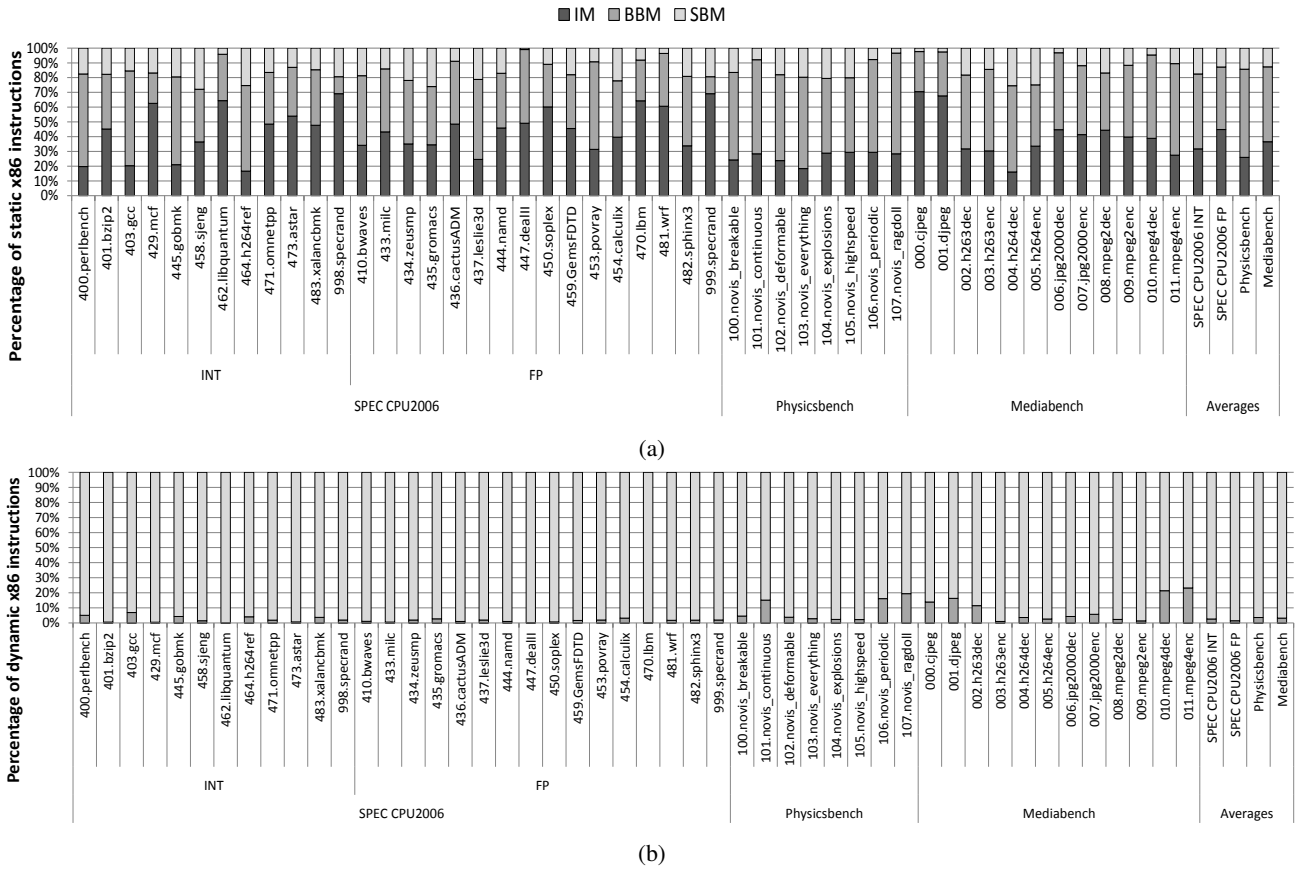
Component	Parameter	Value
General	Issue width	2
Instruction queue	Size	16
Brach predictor	Size of history register	12
L1 I-Cache / L1 D-Cache	Size Block size/Associativity Replacement policy Hit latency	32KB 64B/4 PLRU 1
Stride prefetcher	Number of entries	256
L2 U-Cache	Size Block size/Associativity Replacement policy Hit latency	512KB 128B/8 PLRU 16
Main memory	Hit latency	128
L1 TLB	Entries Replacement policy Hit latency	64/8 way PLRU 1
L2 TLB	Entries Replacement policy Hit latency	256/8 way PLRU 16

The transitional effects are of major importance for HW/SW co-designed processors, as a "heavy" interpreter or translator can lead to major slowdowns. For example, if on average the interpreter consumes 1% of the execution time, by getting a 10 times slower interpreter will result in 10% slowdown. Our experience shows that without careful design a 10x slowdown could easily be the case. Throughout this work, we pay special attention on including these transitional effects in the simulation period. To guarantee this, the simulation period starts from the first instruction of each application. At the same time, in order to avoid these effects being the dominant factor in this simulation period, we simulate a very large number of instructions, 4B (billion) x86 instructions. However, some of our benchmarks have fewer than 4B instructions and as such they run to completion.

The applications used for our studies consist of SPEC CPU2006 [16], Mediabench [17] and Physicsbench [18]. The combination of the three suites provides a rich spectrum of applications for the characterization of TOL and the analysis of the interaction between TOL and the various applications.

III. Quantitative TOL characterization

This section presents a detailed quantitative characterization of the dynamic behavior of TOL. First, we provide information regarding the static and dynamic distribution of the x86 code in the different execution modes of TOL. We then discuss the distribution of the execution time among the various components of TOL and the application code, providing insights on the factors that determine the characteristics of the breakdown. Next, we analyze the performance characteristics of TOL.



(a)

(b)

Fig. 5: x86 code distribution in IM, BBM, SBM: a) Static code; b) dynamic code.

Finally, we examine how the emulated application and TOL interact on the shared resources of the processor and analyze the performance implications of this interaction.

A. Guest code distribution

We study the static and dynamic x86 instruction distribution in the three execution modes of TOL: IM, BM, and SBM. The objective of this analysis is twofold: i) knowing the amount of guest code that will be optimized, as it will affect the overhead generated by TOL; ii) knowing the repetition degree of the optimized code, as it will clarify if the optimization effort can be compensated or not. We assume the following promotion thresholds (analysis not shown due to space limitations): IM/BBth = 5; BB/SBth = 10K.

Figure 5a depicts the static x86 code distribution among the three execution modes. On average, 36% of the static x86 code is not promoted from IM since it is not executed more than 5 times. Moreover, 50% of the code stays is BBM while only 14% is promoted to the SBM. The dynamic x86 code distribution is depicted in Figure 5b. The experimental data show that even with a threshold as high as 10K repetitions, 97% of the dynamic instruction stream comes from the highest level of optimization (SBM), which represents only 14% of the static code. This result motivates using a staged compilation approach, where cold code is filtered and high overhead optimizations with high return of investment are applied only to small fraction of the static code.

In addition, we can extract other interesting insights from Figure 5. First, there is little correlation between the static and the dynamic distribution of the code. For example, *001.djpeg* has 30% of its static code in BBM while the dynamic contribution of that code is 16.24%. However, *462.libquantum* has almost the same static code in BBM (32%) but the dynamic contribution of that code is only 0.1%. The explanation behind this behavior is the repetition factor of the static code, which is strictly application dependent and defined by the high level semantics of the application's algorithm and its input.

The second observation is actually a warning. Although IM and BBM have rather low contribution to dynamic code, TOL still needs to provide a lightweight interpreter/translator that generates good quality code, otherwise they will affect significantly the execution time. A costly interpreter will exacerbate the impact of the small amount of interpreted instructions in execution time. Moreover, from our experience, BBM simple optimizations, like dead code elimination and constant propagation, should be applied. Avoiding those optimizations will impact significantly the distribution of the execution time.

B. Execution time distribution

In HW/SW co-designed processors, the execution time is mainly divided into two categories: i) time spent executing

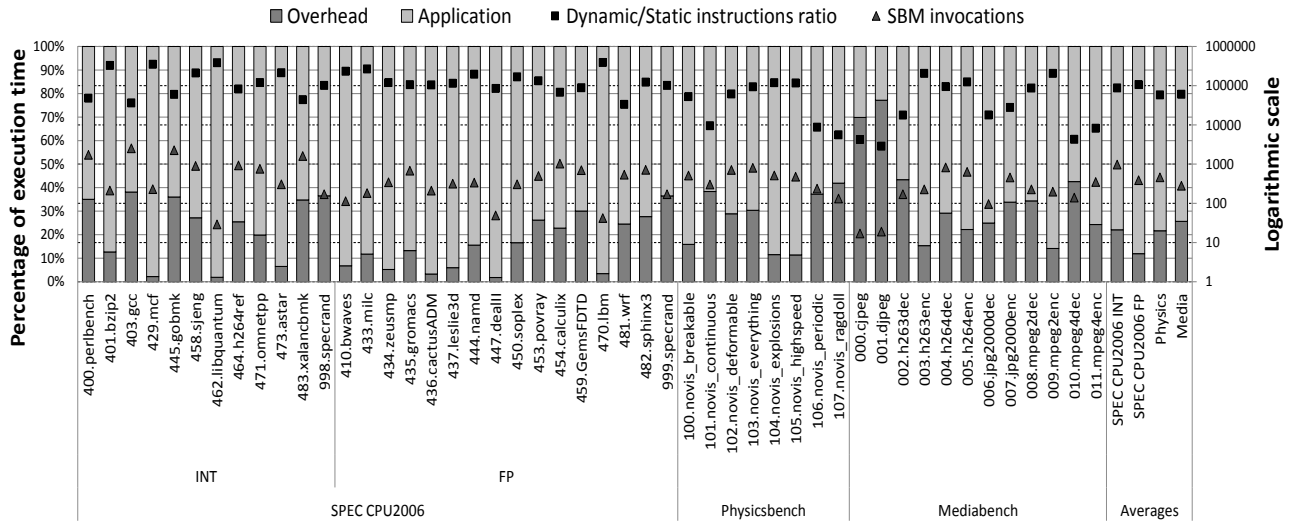


Fig. 6: Breakdown of execution time into TOL and the application.

code corresponding to the guest application, which allows to make forward progress; ii) overhead introduced by TOL while interpreting/translating/optimizing guest instructions, managing the code cache, and performing bookkeeping activities such as collecting profiling information. Note that although with interpretation the application makes forward progress, we consider the interpreter as overhead due to the high cost for emulating one x86 instruction. This assumption has very small impact on the results. Derived from Figures 6 and 7, the global amount of cycles spent in IM is on average 2-3%.

The execution time breakdown is depicted in Figures 6 and 7, where each bar corresponds to a different benchmark. In Figure 6 time is divided into TOL and the application — that is, cycles executing RISC instructions that emulate the guest x86 application. Figure 7 represents the TOL part in Figure 6, which is divided into its major components. The TOL breakdown is as follows (from top to bottom): 1) Code$ lookup: cycles searching for a translation in the code cache; 2) Chaining: cycles connecting BBs or SBs together; 3) SBM: cycles forming and optimizing SBs; 4) BBM: cycles forming and optimizing BBs; 5) IM: cycles interpreting; 6) TOL others: cycles that do not fit in the other categories, e.g. initialization, entry/exit points, control flow within the main execution loop, etc. On average, the overall overhead introduced by TOL in DARCO (Figure 6) is 28% for Mediabench, 22% for Physicsbench, 22% for SPEC INT, and 12% for SPEC FP.

Applications with high dynamic/static instruction ratio (Figures 6) have lower TOL overhead. Since there is high repetition in the code, the overhead introduced by TOL is amortized better. A characteristic example is *462.libquantum*, which has a ratio of 385K repetitions per instruction. On the other hand, applications with low code repetition (i.e. low dynamic/static instructions ratio), like *000.cjpeg* and *001.djpeg*, tend to spend relatively more time in the interpreter and, therefore, have high TOL overhead. Note that *000.cjpeg*, *001.djpeg*, and *433.milc* have a similar static code footprint of 15K x86 instructions. However, *000.cjpeg* and *001.djpeg* have significantly less dynamic x86 instructions than the *433.milc*, which justifies the big difference of the visible overheads.

Another important factor is the relationship between the ratio of dynamic/static instructions and the promotion threshold. If the dynamic/static ratio is close to the threshold, then there is high probability of generating and optimizing multiple SBs which will not repeat enough times to amortize the overhead spent for their creation. From the experimental results we can distinguish two different cases: i) the execution is distributed across a large number of BBs; and ii) the execution is concentrated in just a few BBs.

For the first case, consider applications like *002.h263dec* and *007.jpg2000enc* from Mediabench. The SBM is invoked several times, thus creating a large number of superblocks. Since the repetition factor is close to the SBM invocation threshold, the optimized code will not be executed enough times in order to return the investment done by TOL.

For the second case, consider *006.jpeg2000dec*. Although its dynamic/static instruction ratio is close to the threshold, it has significantly less SBM overhead compared to *007.jpg2000enc*. This is due to the fact that for *006.jpeg2000dec* only 96 superblocks are created, whereas for *007.jpg2000enc* TOL created 450 (Figure 6). This is also reflected in Figure 5 which illustrates that *006.jpeg2000dec* has most of the dynamic execution concentrated in a small number of static instructions. This particular example clearly shows that the overhead depends on the repetition of the BBs that already passed the threshold and not on the overall repetition threshold of all the BBs of the application.

The complexity of the control flow with respect to branches (direct or indirect) encountered in the application is also a very important factor for the introduced overhead. Direct branches are easy to handle since the target address is fixed. This allows translations to be easily connected (linking/chaining), in order to reduce the number of invocations of TOL [19].

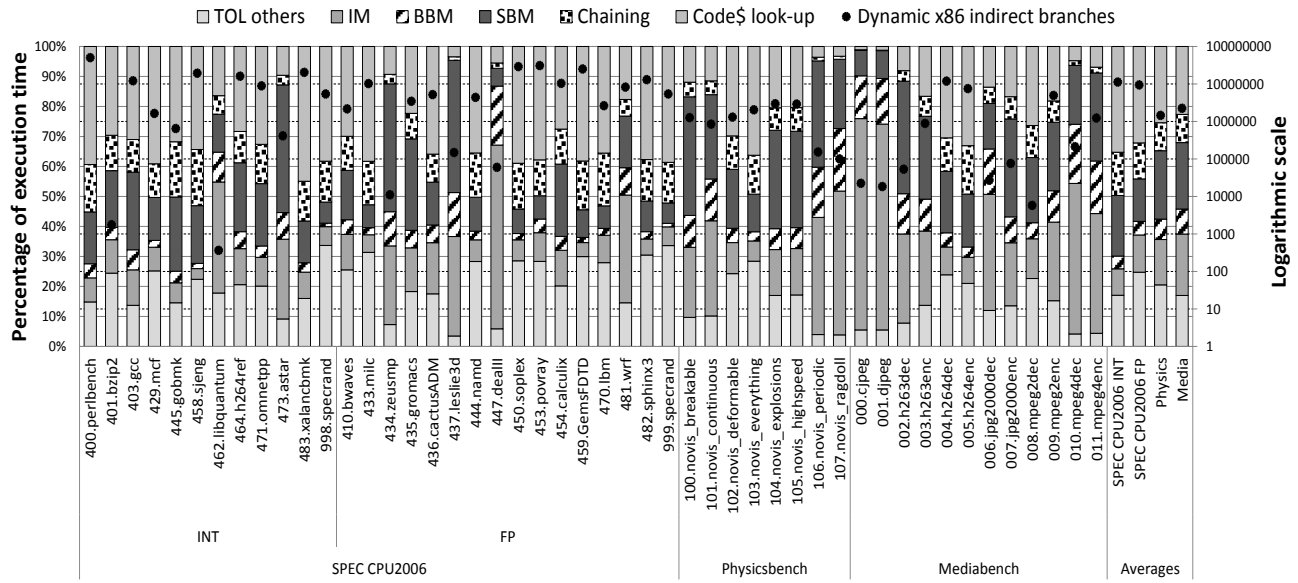


Fig. 7: Breakdown of execution time into the main modules of TOL.

As for the indirect branches, they are way more difficult to handle as their target is determined during their execution. The calculation only gives the guest target address, but it is the responsibility of TOL to identify the corresponding host address by performing a code cache lookup. There is an abundance of related work characterizing the importance of handling the indirect branches in an efficient manner [20], [21], [22]. Currently TOL incorporates the Indirect Branch Translation Cache (IBTC) [20] but, still, the overhead is in the order of tens of RISC instructions.

From the experimental results, we can observe that applications with a high number of indirect branches and returns, like *400.perlbench* and *459.GemsFDTD*, are impacted significantly by the overhead introduced by the code cache lookup. The first time that an x86 indirect branch target is encountered, the execution transitions to TOL in order to perform a code cache lookup, so as to determine the corresponding RISC address that contains the translation of the x86 address (it if was already translated). The IBTC is then updated with the correct pair of x86/RISC addresses, and when the same x86 target address is encountered again, the execution can continue without falling back to TOL. The cost of the transition to TOL is reflected by the "TOL others" part of the bars.

The code cache lookup functions and the transitions to TOL are becoming a major source of overhead for applications with high numbers of x86 indirect branches, e.g. *400.perlbench* (22.7M x86 indirect branches at 4B dynamic x86 instructions), unlike applications with low amount of indirect branches like *401.bzip2* (1933 x86 indirect branches at 4B dynamic x86 instructions) which is verified by previous work [23].

C. TOL performance characteristics

From the point of view of the host processor, TOL is still an application, whose input is the guest x86 application being emulated. In this subsection, we examine the performance characteristics of TOL while emulating the various workloads considered. For the experimental results described in this subsection, we study the execution of TOL in isolation through ignoring in the timing simulator all the instructions that correspond to the emulation of the application.

The performance characteristics of TOL are illustrated in Figure 8. The bars show the RISC instructions per cycle (IPC) executed by TOL. The data cache miss rate, the instruction cache miss rate, and the branch misprediction rate are shown with three types of marks (secondary axis). The most important observation from Figure 8 is that the performance of TOL varies significantly for the different applications. Specifically, the lowest IPC is 0.85 for *445.gobmk* while the highest is 1.48 for *433.milc*. While intuitively TOL was expected to have almost constant performance since it repeats always the same tasks, our experimental results show that this is not true.

The particular characteristics of each application justify this performance variation. The static x86 code of the application is one important contributor. TOL interprets, translates and optimizes the static code, but the effort to do so is defined by the instruction mix. For example, generating code for a *mov reg1,reg2* is cheaper than an *add reg1,reg2* since the latter also modifies the x86 EFLAGS. This also justifies the variation of the branch misprediction rates, since the translator follows different paths, according to the static instruction translated.

The dynamic characteristics of the x86 code are also of major importance. As an example consider *401.bzip2* and *403.gcc*. We know that *401.bzip2* is simpler than *403.gcc* for TOL since it has smaller static code and high repetition, resulting in less SBM invocations for optimization (Figure 6). From the processor's point of view, this implies less memory accesses and fewer branches. Moreover, *403.gcc* has more x86 indirect branches, therefore TOL incurs higher overhead, due

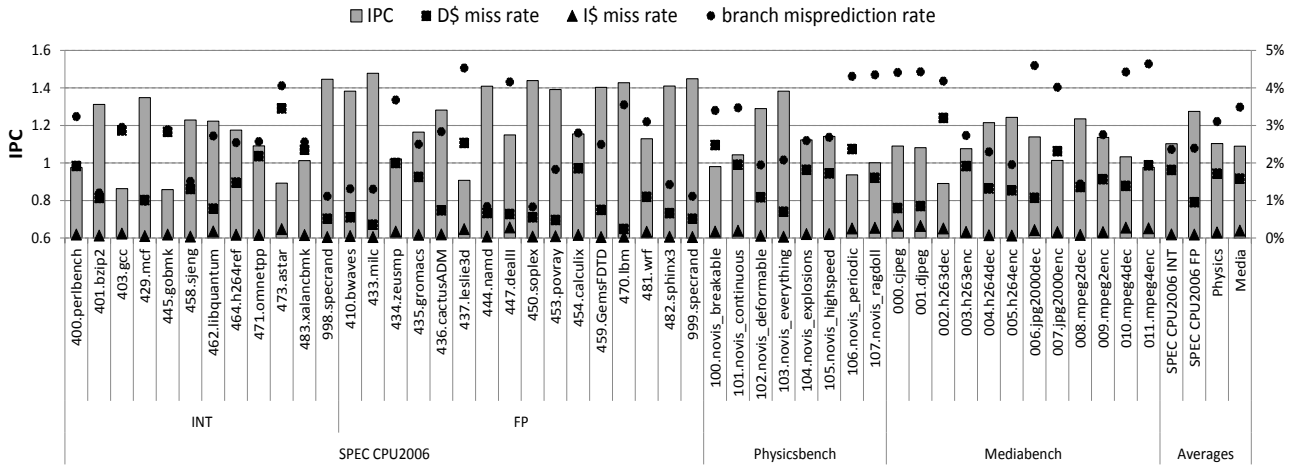


Fig. 8: TOL performance characteristics: IPC, D$ and I$ miss rates and branch mispredictions.

to transitions from the application to TOL and additional code cache lookups. Since the code cache lookup is a data intensive function, the data cache miss rate increases (Figure 8).

Finally, Figure 8 also shows that the impact of the instruction cache on the performance of TOL is negligible. In fact, TOL has small static code footprint which allows the majority of its code to reside in the first level instruction cache.

D. Interaction

In this subsection, we analyze the effect when TOL and the emulated x86 application share the processor resources. Even though they could run in different cores, we want to know the degree of interaction between them. Therefore, if they are forced to run in the same core, we know in advance the penalty we can expect. Due to space limitations, we limit the results to the average of each suite and four special cases, which were selected based on the characteristics described in Section III-B, that is: *470.lbm* has high dynamic/static instructions ratio; *007.jpg2000enc* has a dynamic/static instructions ratio close to the SBM promotion threshold and high SBM activity; *107.novis_ragdoll* has low dynamic/static instructions ratio and high interpreter activity; and *400.perlbench* has high number of x86 indirect branches and transitions to TOL.

Figure 9 depicts the distribution of the execution time, where cycles are categorized into the main sources of bubbles and those spent retiring instructions, from bottom to top: 1) Data cache miss bubbles: caused every time a data access misses; 2) Instruction cache miss bubbles: caused in the back-end, because of an instruction miss; 3) Branch bubbles: propagated to the back-end, because of branch miss predictions; 4) Instruction Scheduling: time during which the IQ does not issue instructions, due to data dependencies or execution unit availability; 5) Instructions: cycles spent retiring instructions. Note that Figure 9 differentiates the percentage that corresponds to TOL and the emulated application.

On average, bubbles account for the 48% of the execution time, of which 26% is spent waiting for data (data cache miss bubbles), 6% is spent with instruction misses that propagated to the back-end, 4% corresponds to branch bubbles, and finally 12% of the time the IQ cannot issue an instruction due to data dependencies or resource unavailability.

It is interesting noting that instructions and bubbles distribute differently between TOL and the application in the four special cases. Applications with high dynamic/static instructions ratio, like *470.lbm*, amortize almost all the overhead introduced by TOL; actually for them the perceived TOL overhead ranges from 2% to 5%. As Figure 9 shows, TOL has minimal impact on the final performance. On the other hand, for applications with low ratio and high IM activity, like *107.novis_ragdoll*, TOL impacts performance significantly since it is responsible for most of the bubbles. The same applies for applications where there is high SBM activity, like *007.jpeg2000enc*, since the overhead is not being amortized by frequent repetition of the optimized code. Finally, applications with frequent transition to TOL because of indirect branches, like *400.perlbench*, are impacted significantly.

The overall performance is impacted by the interaction of TOL and the application on the resources of the processor. Information kept in the data and instruction caches, as well as the branch predictor and the prefetcher is based on temporal and spatial locality. When both entities compete for resource allocation, the information that was built for the one is being polluted by the other resulting in performance degradation. As such, studying the interaction between the application and TOL on the processor is of significant importance. Note that TOL performance depends on the application characteristics, a different case of SMT (Simultaneous Multithreading) processes where the two workloads are independent.

Figure 10 shows the results of this interaction, by presenting the effect of resource sharing. In order to isolate the performance of the application, we ignore the instruction stream of TOL in the timing simulator, thus devoting all resources to the application. We repeat the same for TOL ignoring all the instructions of the application. The results are presented in Figure 10 in terms of relative execution time, i.e. execution time when interaction is not modeled

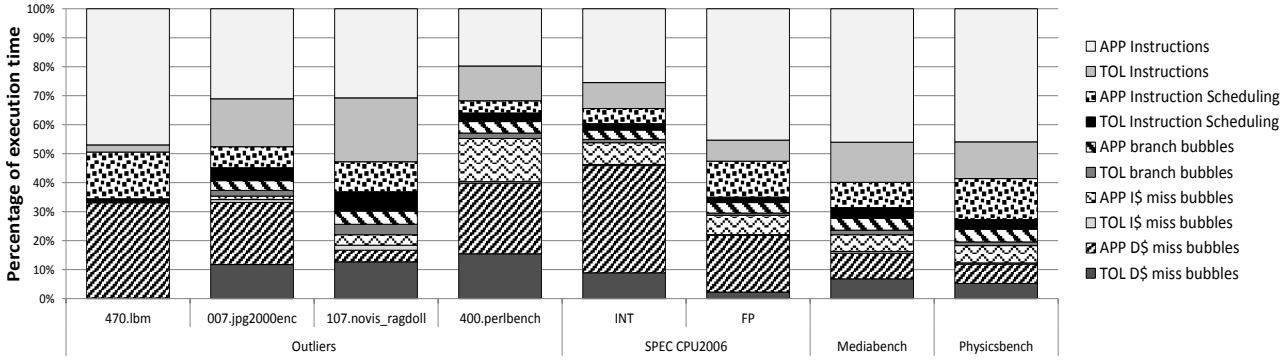


Fig. 9: Main bubble source divided into TOL and the application.

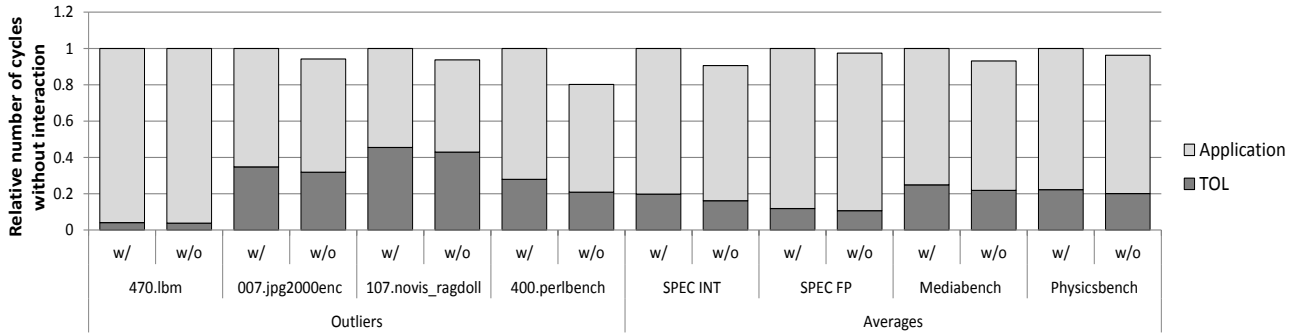


Fig. 10: Performance difference when TOL and Application do not interact.

(w/o) against the execution time when interaction is modeled (w/). It presents this relative execution time for TOL and the application, including data for the average of each suite and the four most relevant outliers.

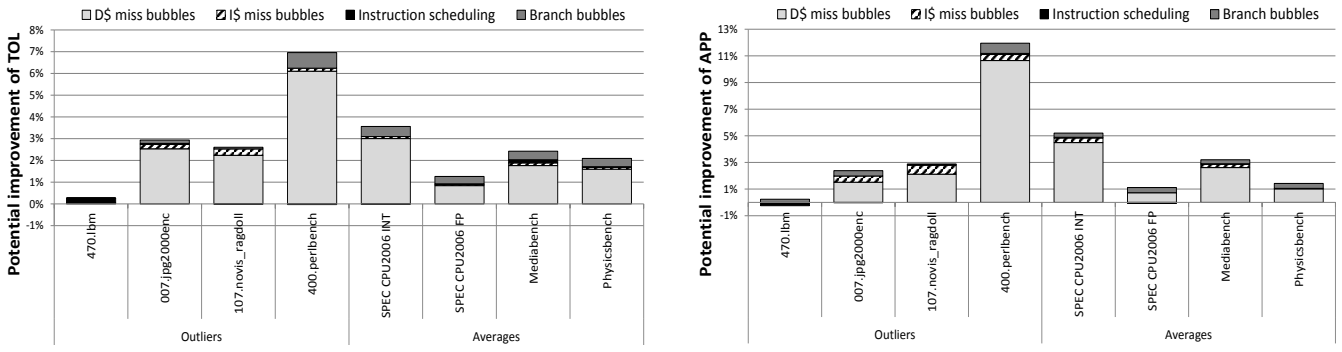
The experimental results clearly depict that the interaction between TOL and the application has a negative impact on performance. In the case of SPEC INT the degradation is 10% on average with TOL contributing 4.2% and the application 5.8%. For SPEC FP the degradation in performance is around 3%, lower than SPEC INT. This can be justified by the low contribution of TOL in the total amount of instructions and bubbles, as shown in Figure 9. Since TOL has lower activity for SPEC FP, compared to SPEC INT, the resources are devoted to the application's emulation for most of the time, thus reducing the effect of the interaction.

It is important to note that the four special categories also follow a trend. For applications like *470.lbm*, with low TOL activity, the performance impact due to the interaction is negligible. Furthermore, for applications with high IM, BBM and SBM activity, like *007.jpg2000enc* and *107.novis_ragdoll*, the impact on performance is around 6% which is aligned with the average for their respective suites. However, for applications with high TOL activity throughout the execution, like *400.perlbench*, performance is degraded significantly, by 20%. Another application in the same category is *403.gcc*, where performance degradation is 14%. The high number of indirect branches in both applications, is causing frequent transitions to TOL for code cache lookups. Since the code cache lookup is a data intensive process, a huge impact on the number of data cache miss bubbles is expected, for both TOL and application.

Figure 11 illustrates the reasons behind the performance degradation. Specifically, it points out how much resources would be affected by the interaction and the potential performance gains, if this interaction was eliminated.

The data cache is the component that could provide the best performance improvement. The case of *400.perlbench* is quite interesting. The frequent transitions between TOL and the application is causing some interesting side effects. Every time TOL is invoked to provide the next region of code to be executed, a code cache lookup is performed. The code cache lookup traverses a table that maps x86 instruction pointers to the position in the code cache where the translation is stored. This data intensive process is evicting lines that are useful to the application, thus increasing the amount of bubbles. While the application is executing, the lines of the lookup tables could be evicted. The impact of this ping-pong effect on the data lines is reflected into 7% data miss bubbles for TOL and 10.6% data miss bubbles for the application.

The impact of other components, while not as high as the data cache, is not negligible. Branch bubbles are increased when the two entities share resources. This can be caused by the pollution of the branch history collected for the one while the other is executing. Instruction cache misses are also increased by the interaction, as instruction lines of TOL can be evicted by the application and vice versa.



(a) (b)

Fig. 11: Potential performance gains by improving the interaction: a) for TOL and b) for the application.

E. Discussion

Our study has identified the key elements that should be considered to improve any HW/SW co-design processor:

- The *data cache* is the main problem, making techniques such as software prefetching, data speculation and opportunistic memory disambiguation of major importance.
- Techniques targeting *instruction scheduling* are also critical for performance. These could include different scheduling algorithms and code reordering.
- The third most important category is the *instruction cache* misses, where TOL could help with software prefetching and code placement in the code cache.
- Finally, there is still potential for software enhancement of *indirect branches*, e.g. by decreasing their number through speculative resolution (using compensation code when speculation was wrong), or as discussed in [24].

It is worth noting that the absolute numbers presented in the study (such as IPC, bubble source distribution, etc) may vary for different guest/host ISAs or microarchitectural parameters. However, the trends observed (such as higher dynamic/static instruction ratio leads to lower TOL overhead) and conclusions made will still hold. The objective of the paper is to identify the critical issues that need to be taken into account when designing HW/SW co-design processors.

IV. Related work

A lot of work has been performed in the field of dynamic binary translation and optimization. In this section we provide a non-exhaustive representative subset of few projects.

Nvidia Denver [7] and Transmeta's Crusoe [1], [2] and Efficeon [3] are the most representative examples of commercial products based on HW/SW co-designed architecture. Nvidia Denver executes ARMv8 binaries on 7-wide in-order cores to keep the power budget to minimum, and relies on dynamic binary optimizations for enhancing performance. Similarly, Transmeta Crusoe executes x86 binaries on a VLIW architecture and employs dynamic optimization to improve performance. The data published in these cases are limited to the power and performance benefits, but no characterization of the processor or the software layer has been made public.

DAISY [4] basically executes PowerPC code on top of a VLIW processor by forming groups from operations among multiple paths. The evaluation focuses on the relative overhead of the dynamic compilation, but no micro-architectural statistics are provided. The BOA project [5] is DAISY's evolution and targets the design of a high frequency processor by keeping the hardware simple. The evaluation focuses on the CPI of the software layer and the emulated application.

PARROT [25] and rePLay [26], [27] are the most representative examples of hardware-based optimizers. The use of a hardware optimizer avoids the translation overheads at the cost of increased transistor count and power, and flexibility loss. PARROT emphasizes on performance and power-awareness, whereas rePLay only on performance. In both cases, they evaluate the quality of their optimizations and present microarchitectural statistics regarding the execution profile.

Purely software based approaches include Dynamo [28], DynamoRIO [29], IA-32 EL [30], and Strata [31]. Dynamo and DynamoRIO are dynamic binary optimizers with same host/guest ISA whereas IA-32 EL translates and optimizes x86 binaries for the Itanium architecture. The published data compare the performance to native execution and present statistics regarding the translation/optimization process. However, they do not present any micro-architectural statistics.

Instead of proposing a complete system, other works focus on the characterization of particular solutions. Examples of works targeting control flow handling include [32], [23], [20], [21], [22]. Code cache management is discussed in [33].

Although for each of the previous systems the results are sound and robust, a global study about the overhead generated by the software layer of a co-designed processor cannot be extracted. To the best of our knowledge, this is the first paper that provides a detailed microarchitectural characterization of the software layer and a detailed study of the interaction between the software layer and the emulated application.

V. Conclusion

This work presented a detailed quantitative characterization of the software layer of a HW/SW co-designed processor. The key observations can be extrapolated to similar systems.

We concluded that the overhead introduced by TOL and hence its performance depends on the characteristics of the emulated application. Specifically, the most important factors are the size of the static code, the dynamic/static instructions ratio and its closeness to promotion threshold, and the complexity of the control flow. TOL demonstrates good performance if the application has small static code and high repetition. However, TOL has lower performance for applications with complex control flow with respect to indirect branches, due to the high amount of code cache lookups performed. We also analyzed the impact of the interaction of TOL and the application on the shared resources of the processor. We observed that the impact of this interaction ranges from almost 0% to 20%, again depending on the characteristics of the emulated application. Finally, we identified four key elements (data cache, instruction scheduling, instructions cache, and indirect branches) that need to be further investigated/analyzed to shrink the variations in TOL performance for the benchmarks analyzed.

ACKNOWLEDGMENT

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness and Feder Funds under grant TIN2013-44375-R.

REFERENCES

[1] A. Klaiber, "The technology behind Crusoe processors," Transmeta Corporation White Paper, Jan. 2000.

[2] J. C. Dehnert, B. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The transmeta code morphing - software: Using speculation, recovery, and adaptive retranslation to address real-life challenges," in *International Symposium on Code Generation and Optimization (CGO)*, March 2003.

[3] K. Krewell, "Transmeta gets more efficeon," Micro-processor Report 17(10), 2003.

[4] K. Ebcioğlu and E. R. Altman, "Daisy: Dynamic compilation for 100architectural compatibility," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, 1997.

[5] S. Sathaye, P. Ledak, J. Leblanc, S. Kosonocky, M. Gschwind, J. Fritts, A. Bright, E. Altman, and C. Agricola, "Boa: Targeting multi-gigahertz with binary translation," in *In Proc. of the 1999 Workshop on Binary Translation*, 1999.

[6] *Intel HW/SW co-designed processor project.*, http://www.eetimes.com/document.asp?doc_id=1266396.

[7] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman, "Denver: Nvidia's first 64-bit arm processor," *IEEE Micro*, vol. 35, no. 2, Mar 2015.

[8] N. Neelakantam, D. R. Ditzel, and C. Zilles, "A real system evaluation of hardware atomicity for software speculation," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV, 2010.

[9] R. Kumar, A. Martnez, and A. González, "Speculative dynamic vectorization to assist static vectorization in a hw/sw co-designed environment," in *20th Annual International Conference on High Performance Computing*, Dec 2013.

[10] C. Wang, Y. Wu, and M. Cintra, "Acceldroid: Co-designed acceleration of android bytecode," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, Feb 2013.

[11] M. Lupon, E. Gibert, G. Magklis, S. Samudrala, R. Martínez, K. Stavrou, and D. R. Ditzel, "Speculative hardware/software co-designed floating-point multiply-add fusion," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014.

[12] R. Kumar, A. Martínez, and A. González, "Efficient power gating of simd accelerators through dynamic selective devectorization in an hw/sw codesigned environment," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 25:1–25:23, Jul. 2014.

[13] M. A. Laurenzano, Y. Zhang, J. Chen, L. Tang, and J. Mars, "Powerchop: Identifying and managing non-critical units in hybrid processor architectures," in *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, ser. ISCA-43, 2016.

[14] D. Pavlou, A. Brankovic, R. Kumar, M. Gregori, K. Stavrou, E. Gibert, and A. Gonzalez, "Darco: Infrastructure for research on hw/sw co-designed virtual machines," *In Proceedings of the 4th Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT11), held in conjuction with ISCA-38*, 2011.

[15] A. Klaiber and S. Chau, "Automatic detection of logic bugs in hardware designs," in *Fourth International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV 2003)*, 2003.

[16] "Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks," http://www.spec.org/cpu2006/, accessed: 2016-04-16.

[17] "MediaBench II Benchmark," http://euler.slu.edu/ fritts/mediabench/, accessed: 2016-04-16.

[18] T. Y. Yeh, P. Faloutsos, S. J. Patel, and G. Reinman, "Parallax: An architecture for real-time physics," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[19] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*, 2005.

[20] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers, "Evaluating indirect branch handling mechanisms in software dynamic translation systems," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2007.

[21] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn, "Vpc prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[22] H.-S. Kim and J. E. Smith, "Hardware support for control transfers in code caches," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003.

[23] B. Dhanasekaran and K. Hazelwood, "Improving indirect branch translation in dynamic binary translators," in *ASPLOS Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments*, ser. RESoLVE, Newport Beach, CA, March 2011.

[24] D. S. McFarlin and C. Zilles, "Bungee jumps: Accelerating indirect branches through hw/sw co-design," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015.

[25] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson, "Power awareness through selective dynamically optimized traces," in *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, June 2004.

[26] S. J. Patel and S. S. Lumetta, "replay: A hardware framework for dynamic optimization," *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 590–608, Jun 2001.

[27] B. Slechta, D. Crowe, B. Fahs, M. Fertig, G. A. Muthler, J. Quek, F. Spadini, S. J. Patel, and S. Lumetta, "Dynamic Optimization of Micro-Operations," in *HPCA*. IEEE Computer Society, 2003.

[28] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00, 2000.

[29] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2003.

[30] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "Ia-32 execution layer: A two-phase dynamic translator designed to support ia-32 applications on itanium®-based systems," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36, 2003.

[31] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '03, 2003.

[32] E. Borin and Y. Wu, "Characterization of dbt overhead." in *IISWC*. IEEE Computer Society, 2009.

[33] K. Hazelwood and M. D. Smith, "Managing bounded code caches in dynamic binary optimization systems," *ACM Trans. Archit. Code Optim.*, vol. 3, no. 3, pp. 263–294, Sep. 2006.